Implementation of a Generic Modular Cryptosystem for the RSA on Reconfigurable Hardware

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

in

Computer Science and Engineering
(Specialization : Information Security)

By

DEEPAK KRISHNANKUTTY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY

ROURKELA, INDIA

2009

Implementation of a Generic Modular Cryptosystem for the RSA on Reconfigurable Hardware

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

in

Computer Science and Engineering
(Specialization : Information Security)

By

DEEPAK KRISHNANKUTTY

UNDER THE GUIDANCE OF

Prof. P. M. Khilar

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY

ROURKELA, INDIA

2009

NATIONAL INSTITUTE OF TECHNOLOGY

ROURKELA

CERTIFICATE

This is to certify that the thesis entitled, **" Implementation of a Generic Modular Cryptosystem for the RSA on Reconfigurable Hardware "** submitted by **Deepak Krishnankutty** in partial fulfillment of the requirements for the award of Master of Technology Degree in Computer Science & Engineering with specialization in **Information Security** during 2007 - 2009 at the National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Date

Prof. P.M. Khilar

Dept. of Computer Science & Engineering

National Institute of Technology

Rourkela-769008

Orissa, India

Acknowledgements

There are many people who have contributed one way or the other to make this thesis work possible. I would like to thank all those people for their assistance.

Special thanks to Professor P. M. Khilar, Professor in the Department of Computer Science Engineering, NIT Rourkela, my advisor for this thesis work, for allowing me to choose such an interesting area of Security in VLSI Design. I truly appreciate and value his encouragement from the beginning to the end of this thesis. His trust and support inspired me in the most important moments for making right decisions and I am glad to work with him.

I am also very thankful to all my classmates and seniors of VLSI lab in the Electronics and Communications Engineering Department, especially Swain Ayas Kanta, K Sudeendra Kumar, Sushant Pattnaik and Dr. Jitendra K Das,my friends Arun Kumar P.S. and George Tom V, who always encouraged me in the successful completion of my thesis work.

I am very thankful to my parents for their continued encouragement and appreciation of the value of my work in the most difficult times of my life.

Deepak Krishnankutty

Contents

Abstract

This report summarizes the work that was initiated from the summer of 2008, on the study and analysis of cryptographic design techniques and their implementation on an *FPGA* board, i.e. the Virtex II pro. The study began with the understanding of a popular HDL language, namely, *Verilog*. Based on the study an implementation of a modular cryptosystem based on the RSA and generic upto a 256 bit modulus was realized. Optimal techniques for developing a high speed RSA cryptosystem is presented in this work.

Through out the thesis the primary tool was the Xilinx based ISE toolkit. However for validation purposes other simulators such as ModelSim was also used. However, the simulations presented in this work utilizes the Xilinx ISE 10.1 Simulator environment. The Xilinx XST 10.1 was used in the synthesis of the implementation.

The division technique utilized a modified non-restoring division scheme. The multiplication scheme used the Karatsuba-Ofman technique. The exponentiation scheme used was the Montgomery Modular exponentiation. The inversion scheme used a modified form of the Extended Euclidean Algorithm which involves no division or multiplication as suggested by Laszlo Hars.

The thesis concludes with suggestions on extending the present implementation of RSA on FPGA.

List of Figures

List of Tables

List of Abbreviations

FPGA : *Field Programmable Gate Array*

VHDL : *VHSIC(Very High Speed Integrated Circuit) Hardware Description Language*

RSA : *Rivest Shamir Adleman public key cryptographic algorithm*

EEA : *Extended Euclidean Algorithm*

BEA : *Binary Euclidean Algorithm*

KOM : *Karatsuba – Ofman Multiplication*

FPGA : *Field Programmable Gate Array*

GF : *Galois Field usually defined over a modulus*

CLB : *Configurable Logic Blocks*

Xilinx ISE : *Xilinx Integrated Software Environment*

Xilinx XPS : *Xilinx Platform Studio for EDK*

EDK : *Embedded Development Kit*

Chapter 1

Introduction

1.1 Introduction

FPGAs are gaining importance both in commercial as well as research settings. The former appreciate the short turn-around times, the lack of Non-reccuring Engineering costs for small volume production and the easy prototyping. Since the technology is far more affordable than custom-manufactured ASICs, even smaller companies can take advantage of the capabilities of large-scale integration.The choice of reconfigurable logic as a target platform for cryptographic algorithm implementations appears to be a practical solution for embedded systems and high-speed applications [1]. Reconfigurable hardware devices are usually distributed in a large geographic area and operated over public networks, making on-site configuration inconvenient or infeasible. Therefore, robust security mechanisms for remote control and configuration are highly needed.

The RSA algorithm is a secure public key algorithm if the modulus size is sufficiently large. It can be used in these applications as a method of exchanging secret information such as keys and producing digital signatures. However, the RSA algorithm is very computationally intensive, operating on very large integers. The RSA algorithm has been adopted by many commercial software products and is built into current operating systems by Microsoft, Apple, Sun, and Novell. Commercial Application Specific Standard Products (ASSPs) like the security processors offered by several vendors have a much higher RSA performance than software implementation [2].

In this thesis, the objective was to design a generic high frequency version of the RSA with the largest possible modulus on the Virtex II pro FPGA using the Verilog HDL. In this regard, a generic cryptosystem with the capabilities of handling a 256 bit modulus was synthesised with some of the best algorithms available for the intrinsic arithmetic operations involved, such as generic multiplication with a 512 bit product output, 256 bit modular exponentiation, 256 bit non-restoring division for the generation of modulus from a divident of size upto 512 bits and 256 bit modular inversion.This thesis focuses primarily on cryptosystems which involve modular arithmetic. Although a key generation component was implemented, the 128 bit prime keys used in the generation of a 256 bit modulus was assumed to be known prior to the implementation. This is because the implementation of the multiplication hardware (recursive Karatsuba-Ofman [3]) was too large and occupied around 40 % of the available slices in the Virtex II pro FPGA. Hence the implementation of an RNG (Random Number Generator) hardware as well a primality tester was excluded for compactness.

Based on the synthesis results, a maximum frequency of 62 Mhz was achieved for the encryption component(exponentiation) in the cryptosystem. The encryption component involves the most computationally intensive part which is modular exponentiation.

1.2 Motivation

Attacks that involve multiple parts of a security system are difficult to predict and model. If cipher designers, software developers and hardware engineers do not understand or review each other's work, security assumptions made at each level of a system's design may be incomplete or unrealistic. As a result security faults often involve unanticipated interactions between components designed by different people. The better solution would be to distribute the levels of security among various levels in a complete system. In this regard a notion of Trust can be achieved from the lower levels to the higher levels.

Trusted Computing [4] more or less serves the benefit of developing models of trust at lower levels of a system [5]. As the range of applications related to reconfigurable logic widens, there occurs the need for secure platforms and secure transactions in such systems. One of the most common public-key algorithms in use for secure transactions is the RSA. Hence the goals of designing a cryptosystem for the RSA on an FPGA become relevant. Implementing the RSA on reconfigurable hardware with a large modulus depends on trade offs between using a high speed algorithms which consume a large amount of space and low speed ones that

1.3 Thesis Layout

The Thesis layout has been structured with a view to provide an understanding of the complexity of the submodules involved in the cryptosystem and to progressively build a complete system that utilizes each of the submodules involved. Based on this view, the various chapters have been presented as follows.

Chapter 2 gives an introduction to the concepts of the RSA algorithm. A review of the previous implementations of the RSA is also discussed. Further, a layout of the RSA implementation also presented.

Chapter 3 introduces the division algorithms that could be utilized for the extraction of modulus in hardware. The implementation of a variant of the non-restoring division algorithm is presented. The synthesis results are compared with earlier implementations.

Chapter 4 introduces the methods by which inversion upon a modulus can be implemented on hardware.The implementation of an inversion technique [6] is also presented. The synthesis results are compared with other variants of the Extended Euclidean Algorithm that was implemented in previous works

Chapter 5 introduces multiplication algorithms that are hardware compatible. The implementation of a recursive Karatsuba-Ofman multiplication with a product result of 512 bits is also presented. The synthesis and simulation results are compared with previous implementations

Chapter 6 introduces various exponentiation techniques. The implementation of the montgomery method of exponentiation is presents. The synthesis results are compared with previous implementations. This version of the montgomery exponentiation employes the direct Karatsuba-Ofman implementation for a higher throughput result.

The last chapter Chapter 7 presents the overall implementation results and summarizes the synthesis details. Further, possible extensions to the current work is also presented.

Chapter 2

The RSA Algorithm

2.1 Introduction

The RSA algorithm was invented by Rivest, Shamir, and Adleman [7]. Let p and q be two distinct large primes.The modulus n is the product of these two primes: n=pq.
Euler's totient function of n is given by

$$\phi(n) = (p-1)(q-1), \tag{2.1}$$

Now, select a number $1 < e < \phi(n)$ such that

$$gcd(e, \phi(n)) = 1, \tag{2.2}$$

and compute d with

$$d = e^{-1} \pmod{\phi(n)} \tag{2.3}$$

using the Extended Euclidian algorithm [8]. Here e is the public exponent and d is the private exponent. Usually one selects a small public exponent e.g., $e = 2^{16} + 1$.The modulus n and the public exponent e are published.The values of d and the prime numbers p and q are kept secret. Encryption is performed by computing

$$C = M^e \pmod{n} \tag{2.4}$$

where M is the plaintext such that $0 \leq M < n$.The number C is the ciphertext from which the plaintext M can be computed using

$$M = C^d \pmod{n} \tag{2.5}$$

The correctness of the RSA algorithm follows from Euler's theorem: Let n and a be positive, relatively prime integers.Then

$$a^{\phi(n)} = 1 \pmod{n}. \tag{2.6}$$

Since we have $ed = 1 \pmod{\phi(n)}$, i.e.,ed = 1 + K$\phi(n)$ for some integer K, we can write

$$\begin{aligned} C^d &= (M^e)^d \pmod{n} \\ &= M^{ed} \pmod{n} \\ &= M^{1+K\phi(n)} \pmod{n} \\ &= M \cdot (M^{\phi(n)})^K \pmod{n} \\ &= M \cdot 1 \pmod{n} \end{aligned} \tag{2.7}$$

provided that gcd(M,n) = 1. The exception gcd(M,n)>1 can be dealt as follows. According to Carmicheal's theorem

$$M^{\lambda(n)} = 1 \pmod{n} \tag{2.8}$$

where $\lambda(n)$ is Carmicheal's function which takes a simple form for $n = pq$, namely,

$$\lambda(pq) = \frac{(p-1)(q-1)}{gcd(p-1,q-1)} \tag{2.9}$$

Note that $\lambda(n)$ is always a proper divisor of $\phi(n)$ when n is the product of distinct odd primes; in this case $\lambda(n)$ is smaller than $\lambda(n)$. Now, the relationship between e and d is given by

$$M^{ed} = M \pmod{n}, if\ ed = 1 \pmod{\lambda(n)}$$

Provided that n is a product of distinct primes, the above holds for all M, thus dealing with the above-mentioned exception gcd(M,n) > 1 in Euler's theorem.

As an example, we construct a simple RSA cryptosystem as follows: Pick p = 11 and q = 13, and compute

$$\begin{aligned} n &= p \cdot q &&= 11 \cdot 13 &&= 143, \\ \phi(n) &= (p-1) \cdot (q-1) &&= 10 \cdot 3 &&= 120. \end{aligned}$$

We can also compute Carmicheal's function of n as

$$\lambda(pq) = \frac{(p-1)(q-1)}{gcd(p-1,q-1)} = \frac{10 \cdot 12}{gcd(10,12)} = \frac{120}{2} = 60$$

The public exponent e is selected such that $1 < e < \phi(n)$ and

$$gcd(e, \phi(n)) = gcd(e, 120) = 1 \tag{2.10}$$

For example,e =17 would satisfy this constraint. The private exponent d is computed by

$$\begin{aligned} d &= e^{-1} \pmod{\phi(n)} \\ &= 17^{-1} \pmod{120} \\ &= 113 \end{aligned}$$

which is computed using the extended Euclidean algorithm, or any other algorithm for computing the modular inverse. Thus, the user publishes the public exponent and the modulus: (e,n) = (13,143), and keeps the following private: d = 113,p = 11 and q = 13. A typical encryption/decryption process is executed as follows:

$$
\begin{aligned}
&\textit{Plaintext}: && M = 50 && \\
&\textit{Encryption}: && M = M^e && \pmod{n} \\
& && C = 50^{17} && \pmod{143} \\
& && C = 85 && \\
&\textit{Ciphertext}: && C = 85 && \\
&\textit{Decryption}: && M = M^d && \pmod{n} \\
& && M = 85^{113} && \pmod{143} \\
& && M = 50 &&
\end{aligned}
$$

2.2 Computation of Modular Exponentiation

Once an RSA cryptosystem is set up, i.e., the modulus and the private and public exponents are determined and the public components have been published, the senders as well as the recipients perform a single operation for signing,verification,encryption, and decryption. The RSA algorithm in this respect is one of the simplest cryptosystems. The operation required is the computation of $M^e \pmod{n}$, i.e., the modular exponentiation. The modular exponentiation operation is a common operation for scrambling [9].It is used in several cryptosystems [10, 11, 12]. Many recent papers have emphasised the need for a fast modular exponentiation on reconfigurable hardware and have presented implementations of the same [13]. This thesis will review some of the exponentiation techniques in Chapter 6 and compare the present work with the previous material.

2.3 Layout of the RSA implementation on FPGA

The primary structure of the implementation presented in this thesis is shown in the following figure 2.1.

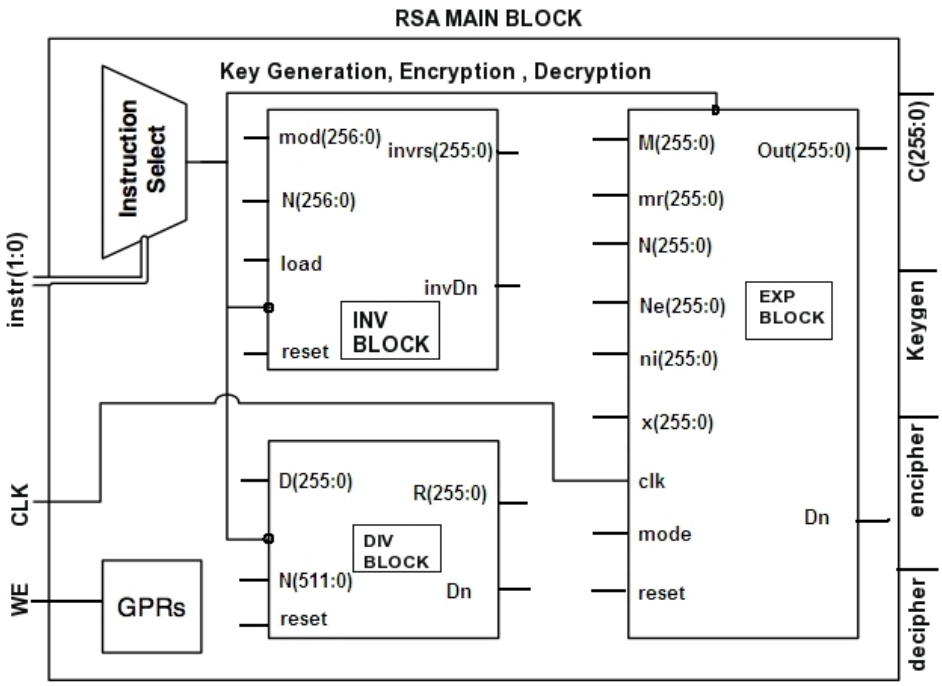

Figure 2.1: Overall Layout of RSA Implementation

The plaintext, the value of encryption exponent e, the values of the 128 bit prime keys p & q for a 256 bit modulus n are expected to be known prior to the implementation in this this layout. The reason for this is that the Montgomery exponentiation design which utilizes the Karatsuba-Ofman multiplier with 512 bit product output, requires 69% of the available slices in the Virtex II pro FPGA which accounts for 9498 slices of the available 13696 slices with a 256 bit modulus. Hence, there leaves space only for implementing the top integration level module and the modules for inversion and modulus retrieval. However, with a view to make the entire cryptosystem efficient, with a high throughput and high speed, we have tried to optimize the modules for consuming the least possible resources available in the FPGA.

The complete system with the RSA integration module takes 90% of the slice count which is 12,374 out of 13,696 available slices in the Virtex II pro, hence producing a large design relative to this FPGA. As a result, an RNG(Random Number Generator), for both generation of prime keys and the value of encryption exponent *e*, and a *primality tester* which would create the prime keys was out of the scope for this implementation. In the above figure 2.1 , we can observe that the montgomery exponentiation module has a dual mode of functionality, i.e., it doubles up as a direct multiplier producing upto 512 bit products of which only a 256 bit product is produced for the value of modulus *n*. This is largely due to the modular structure of Verilog. A module that is instantiated at a top level cannot be shared with a submodule. Hence if the montgomery circuit requires the functionality of simple multiplication, it has to instantiated within the modular montgomery exponentiation circuitry. If resources are to be limited, then there occurs the need for a dual mode functionality.

The inversion module in the above layout utilizes a highly optimized version of the *Extended Euclidean Algorithm* which modifed by Laszlo hars [6]. This algorithm does not require a single division operation nor multiplication operation, thus making it one of the best resource efficient inversion circuits available. The division module which is used for modulus extraction uses a slight enhancement to the original non-restoring division method, whereby the subtraction is performed at a bit level rather than at a word level so as to reduce the hardware resources required for calculation of partial remainders which would have otherwise involved shifting of the *Dividend* by an amount equal to the *Divisor's* bitlength. In this module, subtraction is performed at every bit, once the MSB(Most Significant Bit) of the *Dividend* is detected. The division module returns a 256 bit modulus output for a 512 bit *Divident* input.

2.4 Conclusion

This chapter introduced the RSA algorithm and the modular arithmetic involved in an implementation of the same. The complexity of the cryptosystem's modular exponentiation circuitry was presented. A layout for the entire cryptosystem implemented in this thesis was also presented. The layout briefed the functionality of the modules involved which would set a base for the proceeding chapters.

Chapter 3

Non-restoring Division for Modulus Extraction

3.1 Introduction

The Montgomery exponentiation and RSA require the calculation of the modulus in various steps. The requirement for a modulus calculation for Montgomery exponentiation is mentioned in chapter 6. Although the by products of division include a quotient and a remainder, we are not interested in the quotient; we only need the remainder. Therefore, the steps of the division algorithm can somewhat be simplified in order to speed up the process. The reduction step can be achieved by making one of the well-known sequential division algorithms. In the following sections, we describe the restoring and the non-restoring division algorithms for computing the remainder of t when divided by n. Division is the most complex of the four basic arithmetic operations. Given a dividend t and a divisor n, a quotient Q and a remainder R have to be calculated in order to satisfy

$$t = Q \cdot n + R \; with \; R < n$$

If t and n are positive, then the quotient Q and the remainder R will be positive. The sequential division algorithm successively shifts and subtracts n from t until a remainder R with the property $0 \leq R < n$ is found. However, after a subtraction we may obtain a negative remainder. The restoring and non-restoring algorithms take different actions when a negative remainder is obtained [9].

3.1.1 Restoring Division Algorithm

Let R_i be the remainder obtained during the i^{th} step of the division algorithm. Since we are not interested in the quotient, we ignore the generation of the bits of the quotient in the following algorithm. The procedure given below (3.1) first left-aligns the operands t and n. Since t is 2k-bit number and n is a k-bit number, the left alignment implies that n is shifted k bits to the left, i.e., we start with $2^k n$. Furthermore, the initial value of R is taken to be t, i.e., $R_0 = t$. We then subtract the shifted n from t to obtain R_1; if R_1 is positive or zero, we continue to the next step. If it is negative the remainder is restored to its previous value. In Step 5 of the algorithm, we check the sign of the remainder; if it is negative, the previous remainder is taken to be the new remainder, i.e., a restore operation is performed. If the remainder R_i is positive, it remains as the new remainder, i.e., we do not restore.

The Restoring Division Algorithm
Input: t, n
Output: $R = a \bmod n$
1. $R_0 := t$
2. $n := 2^k n$
3. **for** $i = 1$ **to** k
4. $\quad R_i := R_{i-1} - n$
5. $\quad$ **if** $R_i < 0$ **then** $R_i := R_{i-1}$
6. $\quad n := n/2$
7. **return** R_k

Figure 3.1: Restoring Division Algorithm

The restoring division algorithm performs k subtractions in order to reduce the 2k-bit number t modulo the k-bit number n. Thus, it takes much longer than the standard multiplication algorithm which requires $s = k/w$ inner-product steps, where w is the word-size of the computer.

R_0		101111	001011	t
n		110101		subtract
	−	000110		negative remainder
R_1		101111	001011	restore
$n/2$		11010	1	shift and subtract
	+	10100	1	positive remainder
R_2		10100	101011	not restore
$n/2$		1101	01	shift and subtract
	+	0111	01	positive remainder
R_3		0111	011011	not restore
$n/2$		110	101	shift and subtract
	+	000	110	positive remainder
R_4		000	110011	not restore
$n/2$		11	0101	shift
$n/2$		1	10101	shift
$n/2$			110101	shift and subtract
	+		000010	negative remainder
R_5			110011	restore
R			110011	final remainder

Figure 3.2: Restoring Division Example

In the above example, we give an example of the restoring division algorithm for computing 3019 mod 53, where 3019 = $(101111001011)_2$ and 53 = $(110101)_2$. The result is 51 = $(110011)_2$.

Also, before subtracting, we may check if the most significant bit of the remainder is 1. In this case, we perform a subtraction. If it is zero, there is no need to subtract since $n > R_i$. We shift n until it is aligned with a nonzero most significant bit of R_i. This way we are able to skip several subtract/restore cycles. In the average, k/2 subtractions are performed.

3.2 Non-restoring division implementation

The nonrestoring division algorithm allows a negative remainder. In order to correct the remainder, a subtraction or an addition is performed during the next cycle, depending on the whether the sign of the remainder is positive or negative, respectively. This is based on the following observation: Suppose $R_i = R_{i-1} - n < 0$, then the restoring algorithm assigns $Ri = R_{i-1}$ and performs a subtraction with the shifted n, obtaining

$$R_{i+1} = R_i - n/2 = R_{i-1} - n/2$$

However, if $R_i = R_{i-1}n < 0$, then one can instead let R_i remain negative and add the shifted n in the following cycle. Thus, one obtains

$$R_{i+1} = R_i + n/2 = (R_{i-1} - n) + n/2 = R_{i-1} - n/2,$$

which would be the same value. The steps of the non-restoring algorithm, which implements this observation, are given below:

The Nonrestoring Division Algorithm
Input: t, n
Output: $R = t \bmod n$
1. $R_0 := t$
2. $n := 2^k n$
3. **for** $i = 1$ **to** k
4. **if** $R_{i-1} > 0$ **then** $R_i := R_{i-1} - n$
5. **else** $R_i := R_{i-1} + n$
6. $n := n/2$
7. **if** $R_k < 0$ **then** $R := R + n$
8. **return** R_k

Figure 3.3: Non-restoring Division Algorithm

Note that the nonrestoring division algorithm requires a final restoration cycle in which a negative remainder is corrected by adding the last value of n back to it. In the following we compute 51 = 3019 mod 53 using the non-restoring division algorithm. Since the remainder is allowed to stay negative, we use 2s complement coding to represent such numbers.

R_0	0101111	001011	t
n	0110101		subtract
R_1	1111010		negative remainder
$n/2$	011010	1	add
R_2	010100	1	positive remainder
$n/2$	01101	01	subtract
R_3	00111	01	positive remainder
$n/2$	0110	101	subtract
R_4	0000	110	positive remainder
$n/2$	011	0101	
$n/2$	01	10101	
$n/2$	0	110101	subtract
R_5	1	111110	negative remainder
n	0	110101	add (final restore)
R	0	110011	Final remainder

Figure 3.4: Non-restoring Division Example

In the current implementation, the non-restoring algorithm was modified to suit the minimal resource consumption of the FPGA. Hence, instead of having a large register of 512 bits for both Dividend and the Divisior, the Divisor remains unshifted, however bitwise subtractions have to be performed at every step, which increases the number of subtractions to k where k is the bitlength of the Dividend in the worst case. This is higher than the actual algorithm, however, the actual algorithm shifts a larger number of bits in accordance to the size of the Divisor, making the pre-calculation of *bitlength* for the same mandatory.

3.2.1 Layout for division & Simulation Results

The division algorithm was implemented by means of two modules and the implementation is generic for a Dividend upto 512 bits and a Divisor upto 256 bits. Both the implementations for division and inversion in this work do not make use of a global clock, the looping activity is achieved by means of the switching functionality which is illustrated in figure 3.5 and explained below. This makes it highly resistant to timing based attacks.

The Layout for the division module is shown in figure 3.5.

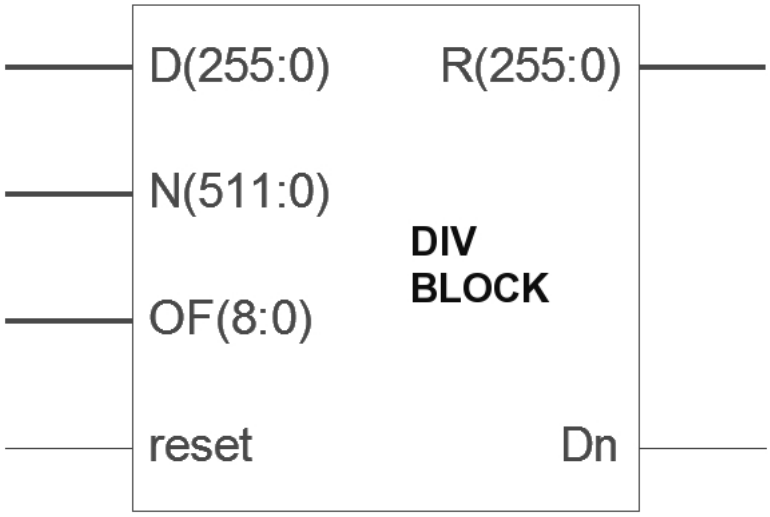

Figure 3.5: Layout of the Division Implementation

The function of each module is described as follows:

1. *Divm* - This submodule serves as a top module for the actual non-restoring step. This module interacts with the submodule *Divloop* to provide a client- server interaction in order to avoid large for loops as seen in the above cases. The *Divm* module additionally uses the computed value of the Divisor bitlength in the OF input.

2. *Divloop* - This submodule only requires individual bits of the 512 bit Dividend to be passed. However, the 2's complement of the Divisor is calculated before in the *Divm* module so it is not recalculated. This value is used for subtraction of partial remainders in each step once the MSB of the Dividend is found. During the entire process, the *Divm* and *Divloop* keep switching until a counter which initially keeps the count of the Dividend's bit length comes to a halt.

A sample simulation of the above is shown in fig 3.6. The simulator used throughout the thesis is the Xilinx ISE XST.

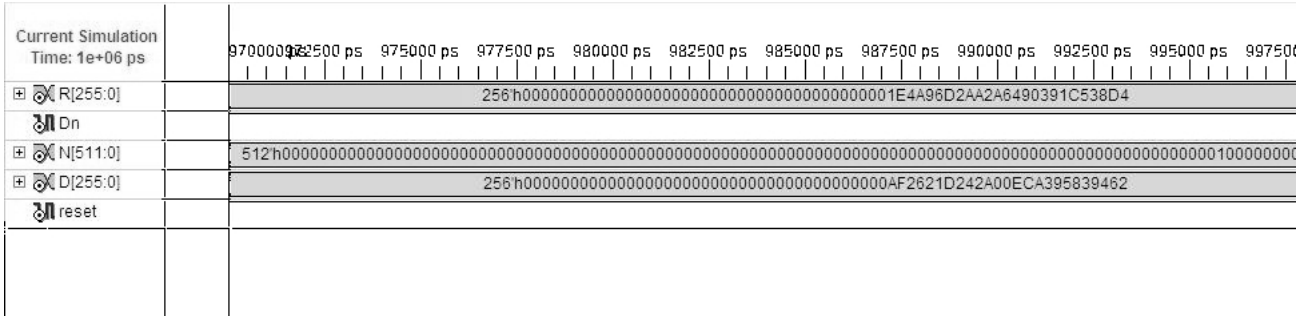

Figure 3.6: Layout of the Division Simulation

In the above simulation, hexadecimal inputs are used. They can be verified as follows,
The Dividend N is first initialized to 586491296780565 in decimal
The Divisor D is first initialized to 1587421 in decimal
The output result is 8dfe3 which is in hexadecimal.
After 20 ns the values are,
The Dividend N is first initialized to 100000000000000000000000000 in hex
The Divisor D is first initialized to af2621d242a00eca395839462 in hex
The output result is 1e4a96d2aa2a6490391c538d4 which is in hexadecimal.
The above results may be further verified using a powerful calculator UBASIC [14].

3.2.2 Synthesis Results

The synthesis results for the implementation of a modified non-restoring division with remainder result of upto 256 bits is shown below in the following table:

FPGA	Logic Slices Used	Combinational Delay(ηs)	Slices Available	Cycles (bits/s)	Throughput
Virtex II pro	455	478	13,696	1	510.75 M

Table 3.1: Synthesis Results for Non-restoring Division

As the above table shows, the *div* module takes very little slice area. Further, the Maximum combinational path delay reported by Xilinx XST was 478 ηs, which makes it a very fast implementation, because of a high throughput rate of $\approx$ 510 Mb/s. Avoiding the use of for loops and substituting it with a combinational feedback switching technique, plays a role in the synthesis of large loops.

No comparison with previous implementations were available because of lack of detailed information in any of the recent work related with FPGAs and non-restoring division.

3.3 Conclusion

The division algorithm used in this work was a non-restoring division version with a few modifications. The layout, simulation results and the synthesis details were shown. The following chapter will discuss the inversion technique.

Chapter 4

Modular Inversion Hardware

4.1 Introduction

Among customary finite field arithmetic operations, namely, addition, subtraction, multiplication and inversion of nonzero elements, the computation of the later is the most time-consuming one. Multiplicative inversion computation of a nonzero element $a \in GF(2^m)$ is defined as the process of finding the unique element $a^{-1} \in GF(2^m)$ such that $a \cdot a^{-1} = 1$. Several algorithms for computing the multiplicative inverse in $GF(2^m)$ have been proposed in literature [15, 16, 17, 18, 19, 20, 21]. In [18], multiplicative inverse is computed using an improved modification of the extended Euclidean algorithm called *almost inverse algorithm*. That iterative algorithm can compute the multiplicative inverse in approximately $2m$ clock cycles [18]. In [Gutub:2002] an architecture able to compute the Montgomery multiplicative inverse for both, $GF(p)$, for a prime p, and $GF(2^m)$ on a unified-field hardware platform was proposed.

Based on Fermat's Little Theorem (FLT) and using an ingenious rearrangement of the required field operations, the Itoh-Tsujii Multiplicative Inverse Algorithm (ITMIA) was presented in [Itoh:1988]. Originally, ITMIA was proposed to be applied over binary extension fields with normal basis field element representation. Since its publication however, several improvements and variations of it have been reported [16, 17, 21, 20], showing that it can be used with other field element representations too.

Unfortunately enough, cryptographic designers have historically shown some resistance to use FLT-related techniques for computing multiplicative inverses when using polynomial basis representation. This phenomenon is probably due to three frequent misconceptions:

1. Computing multiplicative inverses by using FLT-related techniques is inefficient as those methods require many field multiplication and squaring operations;
2. ITMIA is a competitive design option only when using normal basis representation and;
3. The recursive nature of the ITMIA algorithm makes the parallelization of that algorithm rather difficult if not impossible, forcing the implementation of the ITMIA procedure in a sequential manner.

4.1.1 Extended Euclidean Algorithm

Given two polynomials A and B, not both 0, we say that the greatest common divisor of A and B, is the highest polynomial $D = gcd(A,B)$ that divides both A and B. Based on the property $gcd(A,B) = gcd(B \pm (A,A))$, the revered Extended Euclidean Algorithhm (EEA) is able to find the unique polynomials G and H that satisfies Bezout's celebrated formula,

$$A \cdot G + B \cdot H = D,$$

where $D = gcd(A,B)$.

Several variations of the EEA have been proposed in the open literature [1]. EEA variants include: the almost inverse algorithm, first proposed in [22], the Binary Euclidean Algorithm (BEA), the Montgomery inverse algorithm, etc. All those algorithms show a computational complexity proportional to the maximum of A and B polynomial degrees. Algorithm 4.1 shows the binary algorithm as it was reported in [16]. That algorithm takes as inputs the irreducible polynomial P of degree m and the field element A of degree at most $m-1$. It gives as output the field element A^{-1} such that

$$A \cdot A^{-1} = 1 \ (mod\ P).$$

4.1.2 Binary Euclidean Algorithm

The binary GCD algorithm is an algorithm which computes the greatest common divisor of two nonnegative integers. It gains a measure of efficiency over the ancient Euclidean algorithm by replacing divisions and multiplications with shifts, which are cheaper when operating on the binary representation used by modern computers. This is particularly critical on embedded platforms that have no direct processor support for division. The following figure details the BEA, and a brief explanation regarding the functionality of the same is given.

Require: An irreducible polynomial $P(X)$ of degree m, A polynomial $A \in GF(2^m)$.
Ensure: A^{-1} mod $P(x)$.

```
1: U = A; V = P; G = 1; H = 0;
2: while (u ≠ 1 AND v ≠ 1) do
3:    while x divides U do
4:       U = U/x;
5:       if x divides G then
6:          G = G/x;
7:       else
8:          G = (G+P)/x;
9:       end if
10:   end while
11:   while x divides V do
12:      V = V/x;
13:      if x divides G2 then
14:         H = H/x;
15:      else
16:         H = (H+P)/x;
17:      end if
18:   end while
19:   if (deg(U)>deg(V)) then
20:      U = U + V; G = G + H;
21:   else
22:      V = V + U; H = H + G;
23:   end if
24: end while
25: if U=1 then
26:   Return(G);
27: else
28:   Return(H);
29: end if
```

Figure 4.1: Binary Euclidean Algorithm

In steps 4 and 10, the operands U and V are divided by a as many times as possible, respectively. Furthermore, the variables G and H are also divided by X in steps 5-8 and 11-14, respectively. Notice that in case that either G or H are not divisible by a, then an addition with the irreducible polynomial P must be performed first. Eventually, after approximately m iterations, either U or V are equal to 1, which is the condition for exiting the main loop. Either G or H will contain the required multiplicative inverse. The number of iterations, N, required by 4.1 depends on several factors such as design's architecture, target platform and even the exact structure of the irreducible polynomial P(x). N can be estimated as $N \approx m$, where m is the size of the finite field.

4.1.3 Laszlo Hars Modification

This algorithm was proposed by Laszlo Hars in 2006 [6].The original Euclidean GCD algorithm replaces the larger of the two parameters by subtracting the largest number of times the smaller parameter keeping the result nonnegative: $x = x - [x/y] \cdot y$. For this we need to calculate the quotient [x/y] and multiply it with y. In this paper we do not deal with algorithms, which perform division or multiplication. However, the Euclidean algorithm works with smaller coefficients $q \leq [x/y]$, too : $x = x - q \cdot y$. In particular, we can choose q to be the largest power of 2, such that $q = 2^k \leq [x/y]$. The reductions can be performed with only shifts and subtractions, and they still clear the most significant bit of x, so the resulting algorithm will terminate in a reasonable number of iterations. It is well known (see [12]) that for random input, in the course of the algorithm, most of the time [x/y] = 1 or 2, so the shifting Euclidean algorithm performs only slightly more iterations than the original, but avoids multiplications and divisions. See Algorithm 4.2.

```
if (a < m)
  {U ← m; V ← a;
   R ← 0; S ← 1; }
else
  {V ← m; U ← a;
   S ← 0; R ← 1; }
while (||V|| > 1) {
    f ← ||U|| − ||V||
    if (sign(U) = sign(V))
        {U ← U − (V ≪ f);
         R ← R − (S ≪ f); }
    else
        {U ← U + (V ≪ f);
         R ← R + (S ≪ f); }
    if  (||U|| < ||V||)
        {U ↔ V; R ↔ S; }
}
if (V = 0) return 0;
if (V < 0) S ← −S;
if (S > m) return S − m;
if (S < 0) return S + m;
return S; // a^-1 mod m
```

Figure 4.2: Laszlo Hars Shifting Euclidean Algorithm

Repeat the above reduction steps until $V = 0$ *or* ± 1, when $U = GCD(m,a)$. If $V = 0$, there is no inverse, so we return 0, which is not an inverse of anything. (The pathological cases like $m = a = 1$ need special handling, but these do not occur in cryptography.) In the course of the algorithm two auxiliary variables, R and S are kept updated. At termination S is the modular inverse, or the negative of it, within $\pm m$.

4.1.4 Justification for Laszlo Hars Euclidean Algorithm

The algorithm starts with $U = m, V = a, R = 0, S = 1$. If $a > m$, swap (U, V) and (R, S). U always denotes the longer of the just updated U and V. During the course of the algorithm U is decreased, keeping $GCD(U,V) = GCD(m,a)$ true. The algorithm reduces U, swaps with V when $U < V$, until $V = \pm 1$ *or* $0 : U$ is replaced by $U - 2^k V$, with such a k, that reduces the length of U, leading eventually to 0 or ± 1, when the iteration can stop. The *binarylength* $\|U\|$is reduced by at least one bit in each iteration, guaranteeing that the procedure terminates in at most $\|a\| + \|m\|$ iterations.

At termination of the algorithm either $V = 0$ (indicating that $U = 2^k V$ beforehand, and so there is no inverse) or $V = \pm 1$, otherwise a length reduction was still possible. In the later case $1 = GCD(\|U\|, \|V\|) = GCD(m,a)$. Furthermore, the calculations maintain the following two congruencies:

$$U \equiv Ra \pmod{m},\ V \equiv Sa \pmod{m} \tag{4.1}$$

The weighted difference of the two congruencies in (4.1) gives $U - 2^k V \equiv (R - 2^k S) \cdot a \pmod{m}$, which ensures that at the reduction steps (4.1) remains true after updating the corresponding variables : $U = U - 2^k V$, $R = R - 2^k S$. As in the proof of correctness of the original extended Euclidean algorithm, we can see that $\|R\|$ and $\|S\|$ remain less than $2m$, so at the end we fix the sign of S to correspond to V, and add or subtract m to make $0 < S < m$. Now $1 \equiv Sa \pmod{m}$, and S is of the right magnitude, so $S = a^{-1} \pmod{m}$.

4.2 Laszlo Hars Inversion Implementation

4.2.1 Layout of Modular Inversion

The Layout for the inversion module is shown below

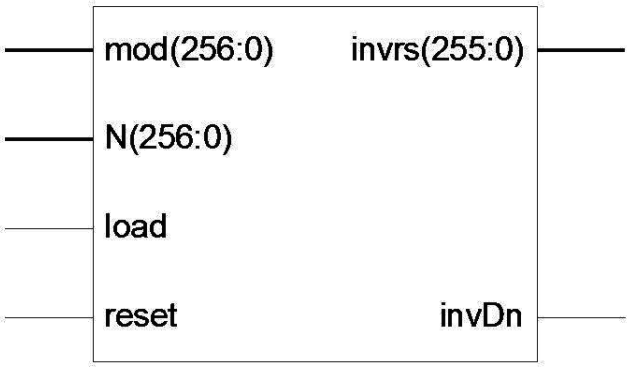

Figure 4.3: Modular Inversion Layout

This module contains a for loop, which when rolled out in a circuit implementation, performs the operation of finding the bitlength. Since, finding the bitlength is pivotal to this module, a switching modulus style of looping was avoided. Ports are described as follows,
Mod takes in a 256 bit modulus ,
N is given a 256 bit Number whose inverse is to be found,
load functions as a 1 bit loading switch which initializes the algorithm 4.2,
reset is a switch which functions as an internal trigger that is used to simulate the while loop in the above algorithm.
invrs contains the result when all the operations are complete.
invDn serves as a notifier to a sequential algorithm above.

4.2.2 Simulation Results

A sample simulation of the above is shown in fig 4.4.

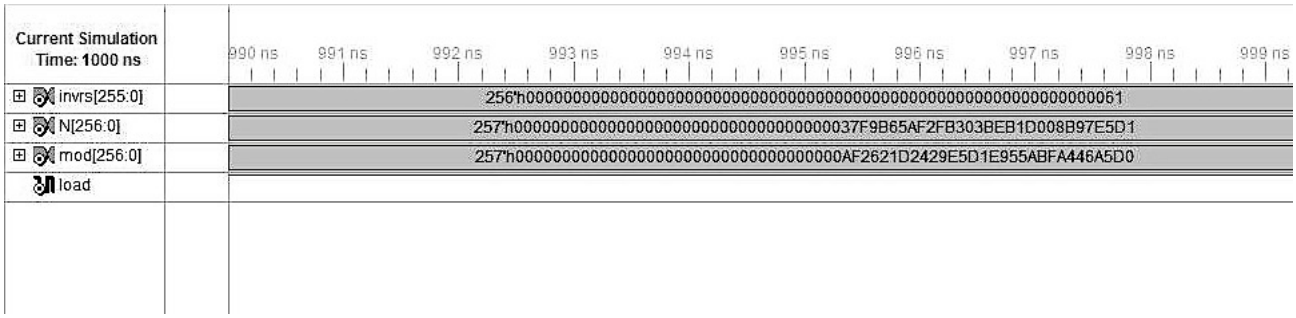

Figure 4.4: Modular Inversion Simulation

The number whose inverse modulo *mod* is to be found,
N was initialized to 290641094883699517630471009136534993 in decimal
The modulus *mod* was initialized to 90942536141028558742437702858851272O in decimal
The output *invrs* was obtained as 61 in hex which is equivalent to 97 in decimal.

4.2.3 Synthesis Results

The synthesis results for the implementation of a modified Extended Euclidean with remainder result of upto 256 bits is shown below in the following table:

Work	*FPGA*	*Slices Used*	*Size*	*Cycles*	*Frequency (Mhz)*	*Timings*
Current work	Virtex II pro	688	256	512(Max)	63	8.192 μS
ITMIA [23]	Virtex II pro	9945	193	40	55	0.724 μS
BEA [23]	Virtex II pro	1195	193	191	76.1	2.509 μS
Parallel ITMIA [24]	Virtex 3200E	12021	193	20	21.2	0.943 μS
ITMIA [20]	Virtex 3200E	1195	193	20	21.2	1.32 μS

Table 4.1: Synthesis Results for Modular Inversion

The comparison shows the benefits of utilizing the algorithm 4.2. The space consumption (CLBs) is definitely lower than other implementations [23, 20] . The Timing details indicate a worst case scenario, where the clock cycles used included 256 bit values for both N and modulus m. Hence if we consider an average value of $N \approx 128bits$ which is the exponent size in RSA, we will have a timing of $\approx 6\mu S$. However, this inversion circuit is used for a dual purpose; it is also used to find the inversion of r in the Montgomery reduction step. However, Itoh-Tsuji inversion circuits have the disadvantage that it can only be used for modulii with powers of 2, hence limiting it's application only to ECCs . Hence, a good space utilization has been achieved in this implementation.

4.3 Conclusion

The inversion algorithm which is used for both Montgomery exponentiation initialization and for finding the decryption exponent in the RSA algorithm was discussed. Although, many variants of the Extended Euclidean Algorithms exist, the author preferred the Laszlo Hars modification because of it's low slice consumption which will be a benefit in building larger RSA systems with a high throughput.

Chapter 5

Modular Multiplication Hardware

5.1 Introduction

The modular exponentiation algorithms perform modular squaring and multiplication operations at each step of the exponentiation. In order to compute $M^e \pmod{n}$ we need to implement a modular multiplication routine. In this section we will study algorithms for computing

$$R = a \cdot b \pmod{n},$$

where a, b, and n are k-bit integers. Since k is often more than 256, we need to build data structures in order to deal with these large numbers. Assuming the word-size of the computer is w (usually w = 16 or 32), we break the k-bit number into s words such that $(s-1)w < k \leq sw$. The temporary results may take longer than s words, and thus, they need to be accommodated as well. In this work, we focus on direct multiplication with the recursive Karatsuba-Ofman implementation, which is very suitable for multi word slice multiplications. However, there is the greatest trade off of space utilization, since this algorithm occupies a large number of slices as will be shown in the synthesis results at the end of this chapter. However, this algorithm was selected as the primary basis for multiplication which would serve the montgomery module above it, and subsequently the Modular exponentiation using the montgomery based square and multiply method which is discussed in the next chapter.

5.2 Modular Multiplication

The problem of modular multiplication and, more specically, the problem of modular reduction has been extensively studied because it is a fundamental building block of any cryptosystem. Among the algorithms that have been proposed we nd:

1. *Sedlaks Modular Reduction* Originally introduced by [25], this algorithm is used by Siemens, in the SLE44C200 and SLE44CR80S microprocessors, to perform modular reduction. Sedlak notices that the algorithm improves the reduction complexity by an average factor of 1/3 when compared to the basic bit-by-bit reduction.

2. *Barrets Modular Reduction* It was originally introduced by [26], in the context of implementing RSA on a DSP processor. Suppose that you want to compute $X \equiv R \pmod{M}$. Then, we can rewrite X as $X = Q \cdot M + R$ with $0 \leq R < M$, which is a well-known identity from the division algorithm [[27], Denition 2.82]. Thus

$$R \equiv X \pmod{M} = X - Q \cdot M \tag{5.1}$$

Barrets basic idea is that one can write Q in (1) as:

$$Q = \lfloor X/M \rfloor = \lfloor \lfloor (X/b^{n-1}) \rfloor (b^{2n}/M)(1/b^{n+1}) \rfloor \tag{5.2}$$

In particular, Q can be approximated by

$$\hat{Q} = Q_3 = \lfloor \lfloor (X/b^{n-1}) \rfloor (b^{2n}/M)(1/b^{n+1}) \rfloor \tag{5.3}$$

Notice that the quantity $\mu = b^{2n}/M$ can be precomputed when performing many modular reductions with the same modulus, as is the case in cryptographic algorithms. Having precomputed μ, the expensive computations in the algorithm are only divisions by powers of b, which are simply performed by right-shifts, and modular reduction modulo b , which is equivalent to truncation.

3. *Brickells Modular Reduction* Originally introduced by [28], is dependent on the utilization of carry-delayed adders and combines a sign estimation technique and Omuras modular reduction [29].

4. *Quisquaters Modular Reduction* Quisquaters algorithm, originally presented by [30], can be thought of as an improved version of Barrets reduction algorithm.The method is used in the Phillips smart-card chips P83C852 and P83C855, which use the CORSAIR crypto-coprocessor [31]. Quisquaters algorithm, as presented in [31], is a combination of the interleaved multiplication reduction method (basically, combine a normal multiprecision algorithm with modular reduction, making use of the distributivity property of the modular operation) and a method that makes easier and more accurate the estimation of the quotient Q in (1).

5. *Montgomery Modular Multiplication* The Montgomery algorithm, originally introduced by [32], is a technique that allows efcient implementation of the modular multiplication without explicitly carrying out the modular reduction step.

5.2.1 Montgomery Modular Multiplication

The idea behind Montgomery's algorithm is to transform the integers in M-residues and compute the multiplication with these M-residues. At the end, one transforms back to the normal representation. As with Quisquaters and Barrets method, this approach is only beneficial if we compute a series of multiplications in the transform domain (e.g., modular exponentiation).

The Montgomery reduction algorithm is as follows:
Given integers M and R with $R > M$ and gcd(M,R)=1,

$$M' = -M \equiv -1 \pmod{R}.$$

Let T be an integer such that $0 \leq T < MR$.
If $Q \equiv TM' \pmod{R}$, then Z=(T+QM)/R is an integer and further,

$$Z \equiv TR^{-1} \pmod{M}.$$

This is just the reduction step involved in a modular multiplication.The multiplication step can be carried out via multiprecision multiplication (see, e.g., [27], Chapter 14). As with previous algorithms, one can interleave multiplication and reduction steps and is further discussed in the next section. The result is shown in Algorithm 5.1. In practice R is a multiple of the word size of the processor and a power of two. This means that M, the modulus, has to be odd (because of the restriction gcd(M,R)=1) but this does not represent a problem as M is a prime or the product of two primes (RSA) in most practical cryptographic applications. In addition, choosing R a power of 2, simplies the computation of Q and Z as they become simply truncated (modular reduction by R) and right shifting (division by R). Notice that $M' \equiv -M \pmod{R}$. In [33] it is shown that if $M = \sum_{i=0}^{n-1} m_i b^i$, for some radix b typically a power of two, and $R = b^n$, then M' can be substituted by $m'_0 = -M^{-1} \pmod{b}$.

Algorithm 5.1 Montgomery Multiplication Algorithm

Require: $X = \sum_{i=0}^{n-1} x_i b^i, Y = \sum_{i=0}^{n-1} y_i b^i, M = \sum_{i=0}^{n-1} m_i b^i$, with $0 \leq X, Y < M, b > 1, m' = -m_0^{-1} \pmod{b}, R = b^n, gcd(b, M) = 1$

Ensure: $Z = X \cdot Y \cdot R^{-1} mod M$

1: $Z \leftarrow 0$\{ where $Z = \sum_{i=0}^{n} z_i b^i$\}

2: **for** i=0 to n-1 do

3: $Z \leftarrow (Z + x_i \cdot Y + q_i \cdot M)/b$

4: $q_i \leftarrow (z_0 + x_i \cdot y_0) m' \pmod{b}$

5: **end for**

6: **if** $Z \geq M$

7: $Z \leftarrow Z - M$

8: **end if**

9: Return(Z)

In [34], the authors simplify the combinatorial logic needed to implement Montgomery reduction.The idea in [34], is to shift Y by two digits (i.e., multiply Y by b^2) and thus, make q_i in Step 4 of Algorithm 5.1 independent of Y . Notice that one could have multiplied Y by b instead of b^2 and have also obtained a q_i independent of Y. However, by multiplying Y by b^2, one gets q_i to be dependent only on the partial product Z and on the lowest two digits of the multiple of M (i.e., $q_i \cdot M$). The price of such a modication is two extra iterations of the for-loop for which the digits of X are zero.

In Mazzeo et al, an architecture and the FPGA implementation of a digit-serial RSA processor was proposed [35]. A Xilinx Virtex-E 2000-8bg560 was used for implementing the proposed architecture.This approach has a trade off with chip area and speed.

5.3 Efficient techniques for direct multiplication

Various methods are available for a high throughput based multiplication. For a bitwise implementation, the booth's multiplication is one of the most basic techniques.

5.3.1 Standard Multiplication Algorithm

Let a and b be two s-digit (s-word) numbers expressed in radix W as:

$$a = (a_{s-1}a_{s-2}\cdots a_0) = \sum_{j=0}^{s-1} a_i W^i,$$

$$b = (b_{s-1}b_{s-2}\cdots b_0) = \sum_{j=0}^{s-1} b_i W^i,$$

where the digits of a and b are in the range [0,W-1]. In general W can be any positive number. For computer implementations, we often select $W = 2^w$ where w is the word-size of the computer, e.g., w = 32. The standard (pencil-and-paper) algorithm for multiplying a and b produces the partial products by multiplying a digit of the multiplier (b) by the entire number a, and then summing these partial products to obtain the final number 2s-word number t. Let t_{ij} denote the (Carry,Sum) pair produced from the product $a_i \cdot b_j$. For example, when W = 10, and a_i = 7 and b_j = 8, then t_{ij} = (5,6). The t_{ij} pairs can be arranged in a table as

				a_3	a_2	a_1	a_0
$\times$				b_3	b_2	b_1	b_0
				t_{03}	t_{02}	t_{01}	t_{00}
			t_{13}	t_{12}	t_{11}	t_{10}	
		t_{23}	t_{22}	t_{21}	t_{20}		
$+$	t_{33}	t_{32}	t_{31}	t_{30}			
t_7	t_6	t_5	t_4	t_3	t_2	t_1	t_0

Figure 5.1: Standard multiplication product

The last row denotes the total sum of the partial products, and represents the product as an 2s-word number. The standard algorithm for multiplication essentially performs the above digit-by-digit multiplications and additions. In order to save space, a single partial product variable t is being used. The initial value of the partial product is equal to zero; we then take a digit of b and multiply by the entire number a, and add it to the partial product t.

The partial product variable t contains the final product $a \cdot b$ at the end of the computation. The standard algorithm for computing the product $a \cdot b$ is given below:

The Standard Multiplication Algorithm
Input: a, b
Output: $t = a \cdot b$
0. Initially $t_i := 0$ for all $i = 0, 1, \ldots, 2s-1$.
1. **for** $i = 0$ **to** $s-1$
2. $\quad C := 0$
3. $\quad$ **for** $j = 0$ **to** $s-1$
4. $\quad\quad (C, S) := t_{i+j} + a_j \cdot b_i + C$
5. $\quad\quad t_{i+j} := S$
6. $\quad t_{i+s} := C$
7. **return** $(t_{2s-1} t_{2s-2} \cdots t_0)$

Figure 5.2: Standard Multiplication Algorithm

In order to implement this algorithm, we need to be able to execute Step 4:

$$(C,S) = t_{i+j} + a_j \cdot b_i + C,$$

where the variables t_{i+j}, a_j, b_i, C, and S each hold a single-word, or a W-bit number. This step is termed as an inner-product operation which is common in many of the arithmetic and number-theoretic calculations. The inner-product operation above requires that we multiply two W-bit numbers and add this product to previous *carry* which is also a W-bit number and then add this result to the running partial product word t_{i+j}. From these three operations we obtain a 2W-bit number since the maximum value is

$$2^W - 1 + (2^W - 1)(2^W - 1) + 2^W - 1 = 2^{2W} - 1.$$

Also, since the inner-product step is within the innermost loop, it needs to run as fast as possible. Of course, the best thing is to have a single microprocessor instruction for this computation. A brief inspection of the steps of this algorithm reveals that the total number of inner-product steps is equal to s^2. Since $s = k/w$ and w is a constant on a given computer, the standard multiplication algorithm requires $O(k^2)$ bit operations in order to multiply two k-bit numbers. This algorithm is asymptotically slower than the Karatsuba algorithm and the FFT-based algorithm which are to be studied next. However, it is simpler to implement and, for small numbers, gives better performance than these asymptotically faster algorithms.

5.3.2 Karatsuba-Ofman Algorithm

We now describe a recursive algorithm which requires asymptotically fewer than $O(k^2)$ bit operations to multiply two k-bit numbers. The algorithm was introduced by two Russian mathematicians Karatsuba and Ofman in 1962 cite. The following is a brief explanation of the algorithm. First, decompose a and b into two equal-size parts:

$$a = 2^h a_1 + a_0,$$
$$b = 2^h b_1 + b_0,$$

i.e., a_1 is higher order h bits of a and a_0 is the lower h bits of a, assuming k is even and$2h = k$. Since we will be worried only about the asymptotics of the algorithm, let us assume that k is a power of 2. The algorithm breaks the multiplication of a and b into multiplication of the parts a_0, a_1, b_0, and b_1. Since

$$\begin{aligned} t &= a \cdot b \\ &= (2^h a_1 + a_0)(2^h b_1 + b_0) \\ &= 2^{2h}(a_1 b_1) + 2^h(a_1 b_0 + a_0 b_1) + a_0 b_0 \\ &= 2^{2h} t_2 + 2^h t_1 + t_0, \end{aligned}$$

the multiplication of two $2h$-bit numbers seems to require the multiplication of four h-bit numbers.The Karatsuba-Ofman algorithm is based on the following observation that, in fact, three half-size multiplications suffice to achieve the same purpose:

$$\begin{aligned} t_0 &= a_0 \cdot b_0, \\ t_2 &= a_1 \cdot b_1, \\ t_1 &= (a_0 + a_1) \cdot (b_0 + b_1) - t_0 - t_2 = a_0 \cdot b_1 + a_1 \cdot b_0. \end{aligned}$$

This yields the Karatsuba-Ofman recursive multiplication algorithm (KORMA) which is illustrated below:

function $KORMA(a,b)$
$t_0 = KORMA(a_0, b_0)$
$t_2 = KORMA(a_1, b_1)$
$u_0 = KORMA(a_1 + a_0, b_1 + b_0)$
$t_1 = u_0 - t_0 - t_2$
return $(2^{2h}t_2 + 2^h t_1 + t_0)$

Let T(k) denote the number of bit operations required to multiply two k-bit numbers using the Karatsuba Ofman algorithm. Then,

$$T(k) = 2T(\frac{k}{2}) + T(\frac{k}{2} + 1) + \beta k \approx 3T(\frac{k}{2}) + \beta k$$

Similarly, βk represents the contribution of the addition, subtraction, and shift operations required in the recursive Karatsuba-Ofman algorithm. Using the initial condition T(1) = 1, we solve this recursion and obtain that the Karatsuba-Ofman algorithm requires

$$O(k^{log_2 3}) = O(k^{1.58})$$

bit operations in order to multiply two k-bit numbers. Thus, the Karatsuba-Ofman algorithm is asymptotically faster than the standard (recursive as well as nonrecursive) algorithm which requires $O(k^2)$ bit operations. However, due to the recursive nature of the algorithm, there is some overhead involved. For this reason, Karatsuba-Ofman algorithm starts paying off as k gets larger. Current implementations indicate that after about k = 250, it starts being faster than the standard nonrecursive multiplication algorithm. Also note that since $a_0 + a_1$ is one bit larger, thus, some implementation difficulties may arise. However, we also have the option of stopping at any point during the recursion. For example, we may apply one level of recursion and then compute the required three multiplications using the standard nonrecursive multiplication algorithm. This is primarily the reason why we chose to stop at a level when the size of individual multiplications reached 17 bits to be precise. This is due to the fact that the internal multipliers in Xilinx based FPGAs can have upto 18 bit by 18 bit multiplication operands.

5.3.3 FFT-based Multiplication Algorithm

The fastest multiplication algorithms use the fast Fourier transform. Although the fast Fourier transform was originally developed for convolution of sequences, which amounts to multiplication of polynomials, it can also be used for multiplication of long integers.

There are many Fourier primes, i.e., primes p for which FFTs in modulo p arithmetic exist. Moreover, there exists a reasonably efficient algorithm for determining such primes along with their primitive elements [31]. From these primitive elements, the required primitive roots of unity can be efficiently computed. This method for multiplication of long integers using the fast Fourier transform over finite fields was discovered by Schönhage and Strassen [45]. It is described in detail by Knuth [19]. A careful analysis of the algorithm shows that the product of two k-bit numbers can be performed using O($k\ logk\ loglogk$) bit operations. However, the constant in front of the order function is high. The break-even point is much higher than that of Karatsuba-Ofman algorithm. It starts paying off for numbers with several thousand bits. Thus, they are not very suitable for performing RSA operations.

5.4 Hybrid Karatsuba-Ofman Implementation

The structure for implementing a hybrid variant of the Karatsuba-Ofman multiplier is shown in the figure 5.3:

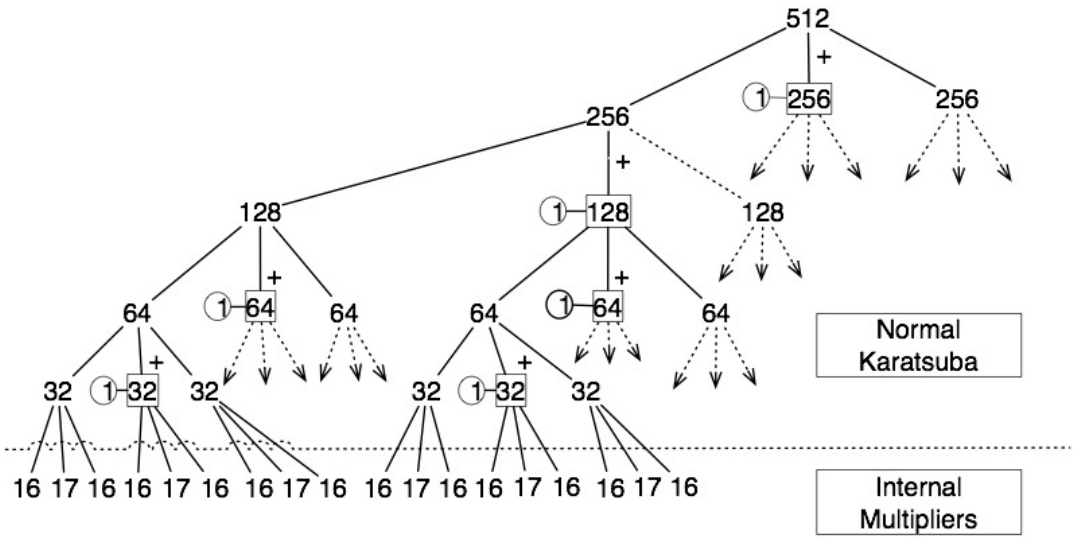

Figure 5.3: Abstract 512 bit Hybrid Karatsuba Structure

The calls that are specially marked indicate an optimization step that generalizes the implementation. The optimization step removes the Most Significant Bit (MSB) from the sum of the parts in the recursive call. The step ensures an even division and a reduction in specialized hardware that would have been necessary if the multiplication steps were sequentially performed. Hence this optimization is beneficial for the sequential version of the recursive karatsuba multiplier. However, only a slight benefit occurs for a parallel implementation. For the bit that was removed, a partial product is added by means of the following observation 5.4. Assume that the addition result is stored into SumX and SumY for the first and second addition operations respectively in the Karatsuba-Ofman Algorithm.

$$
\begin{aligned}
&If\ MSB(SumX) = 1\ and\ MSB(SumY) = 1\ then,\\
&newProduct3 = SumX + SY,\\
&where\ SX\ and\ SY\ are\ SumX\ and\ SumY\ with\ the\ MSB\\
&removed,\ respectively\\
&If\ MSB(SumX) = 0\ and\ MSB(SumY) = 1\ then,\\
&newProduct3 = SumX,\\
&If\ MSB(SumX) = 1\ and\ MSB(SumY) = 0\ then,\\
&newProduct3 = SY,\\
&else\ newProduct3 = 0
\end{aligned}
\tag{5.4}
$$

Observation for MSB bits of the two Sum results in Karatsuba-Ofman

This observation arises, from analysis of the carry bit in the partial products that are formed during any normal multiplication. Thus, the objective of this observation is only to remove the MSB bit from the sum which results initially causes an extra bit during multiplication. If the MSB bits were present in the sum result , then the even-ness of the Karatsuba division is lost. As a result, the extra bit requires a call to an instance of the recursive hardware with extra bits. This carries on to the sub-levels of the implementation. Hence, such a situation was avoided.

5.4.1 Simulation Results

The simulation of a parallel recursive Karatsuba-Ofman technique with the modifications outlined above is given below :

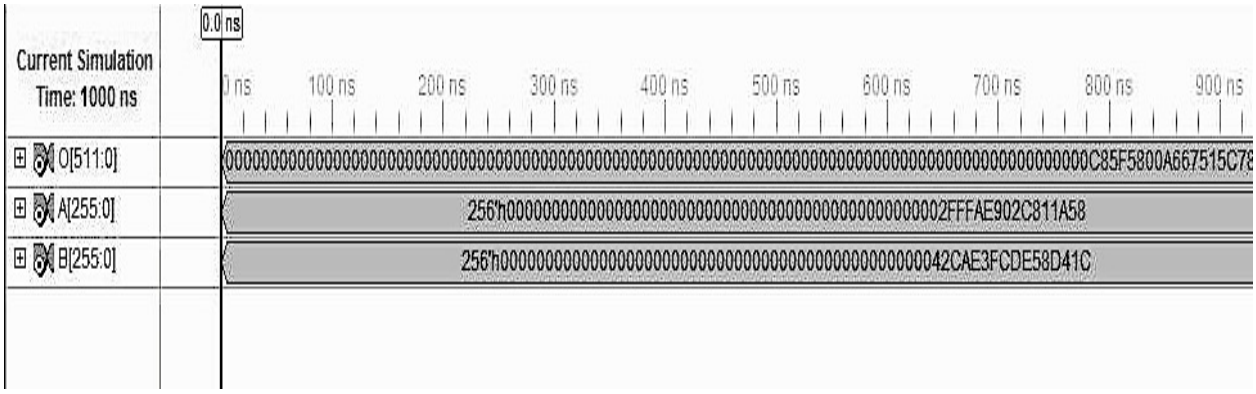

Figure 5.4: Karatsuba-Ofman Multiplier Simulation

A denotes the first operand and *B* , the seccond,
A was initialized to 3458674973089012312 in decimal
B was initialized to 4812909827000423452 in decimal
The output *O* was obtained as c85f5800a667515c78fb479262fc1a0 in hex which is equivalent to 97 in hexadecimal.

5.4.2 Synthesis Results

The synthesis results for the implementation of a modified Extended Euclidean with remainder result of upto 256 bits is shown below in the following table:

Work	*FPGA*	*Slices*	*Size* of Product(bits)	*Cycles*	*Timing*	$\frac{bits}{Slices\ x\ Timing}$
Current work	Virtex II pro	6562	512	1	$61.653\eta S$	1.279M
Current work	Virtex II pro	1887	256	1	$44.047\eta S$	3.0832M
Non-Redundant KOM by [36]	Virtex II pro	5307	163	1	$12.56\eta S$	2.445M
Less Recursive KOM by [36]	Virtex II pro	5409	163	1	$13.37\eta S$	2.254M
Parallel KOM by [36]	Virtex II pro	5840	163	1	$14.73\eta S$	1.895M
KOM by [37]	Virtex II pro	1480	240	30	$378\eta S$	0.429M
Recursive Classical by [37]	Virtex II pro	1582	240	56	$523\eta S$	0.290M
KOM by [38]	Virtex II pro	1660	240	54	$655\eta S$	0.221M

Table 5.1: Synthesis Results for Karatsuba-Ofman Multiplier

We measure efficiency by taking the ratio of number of bits processed over slices multiplied by the time delay achieved by the design, namely,

$$\frac{bits}{Slices\ x\ timings}$$

As is obvious from the above table, this work has achieved a better throughput in terms of $\frac{bits}{Slices\ x\ Timing}$ when the 256 bit product is considered. If the output is raised to 256 bits, then the combinational delay increases proportionately, however the slice count becomes ≈ 5 times the 256 bit version. However, it is most natural of the KOM, because of it's inherent structure. Although implementations in [36], show a good throughput, they have a high slice count. In comparison to the lower slice count version by [37] and [38] , the current work fares better because of it's efficiency. Further, it must be mentioned that this slice count does not indicate the number of internal multipliers used in this implementation.

This implementation utilizes 81 of the 136 internal multipliers in the 512 bit version and 27 multipliers for the 256 bit version. If the slice count conversion for such multipliers would indicate a variable area consumption, then the utilization of available resources on the *FPGA* would be the major factor for this work. The Classical version was included only for comparison purposes.Hence by far, this is a very good implementation.

5.5 Conclusion

Various techniques for implementing modular multiplication were discussed in this chapter, primarily due to the fact that this is one of the major contributor to the slice consumption hardware. The objective of a high-throughput design will be speed. However, this affects the space utilization and subsequent resource consumption. This work has achieved a better efficiency in terms of both slice count and response timings as was discussed. The hybrid version of the Karatsuba-Ofman multiplier which was used in this work was discussed, following which the simulation and synthesis were analysed.

Chapter 6

Modular Exponentiation Hardware

6.1 Introduction

Modular exponentiation can be defined in terms of field multiplication as follows. Let a be a positive integer in [1, n]. Let also e be defined as an arbitrary positive integer. Then, we define modular exponentiation as the problem of finding the number y such that,

$$y = x^e \pmod{n} \tag{6.1}$$

Taking advantage of the linearity property of the modular operation,6.1 can be evaluated by performing a reduction modulo n at each step of the exponentiation thus guaranteeing that all the partial results will not grow larger than twice the length of the modulus. In the rest of this Section we will consider that every multiplication operation always includes a subsequent reduction step.

In general one can follow two strategies in order to optimize the computation of 6.1. One approach is to implement field multiphcation, the main building block required for field exponentiation, as efficiently as possible. This was cover in Chapter 5. The other is to reduce the total number of multiplications needed to compute 6.1. In this Section we address the latter approach, assuming that arbitrary choices of the base x are allowed but considering that the exponent e has been previously fixed.

6.2 Efficient techniques for exponentiation

In this section, we include a brief review of the main deterministic heuristic proposed in the literature for computing field exponentiation [1].

6.2.1 Binary Strategies

Let e be an arbitrary m-bit positive integer e, with a binary expansion representation given as, $e = (1e_{m-2}\cdots e_1e_0)_2 = 2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i$. Then,

$$y = x^e = x^{2^{m-1}+\sum_{i=0}^{m-2} 2^i e_i} = x^{2^{m-1}} \cdot \prod_{i=0}^{m-2} x^{2^i e_i} \tag{6.2}$$

Binary strategies evaluate 6.2 by scanning the bits of the exponent e one by one, either from left to right (MSB-first binary algorithm) or from right to left (LSB-first binary algorithm) applying the so-called Horner's rule. Both strategies require a total of $m-1$ iterations. At each iteration a squaring operation is performed, and if the value of the scanned bit is one, a subsequent field multiplication is performed. Therefore, the binary strategy requires a total of $m-1$ squarings and $H(e)-1$ field multiplications, where $H(e)$ is the Hamming weight of the binary representation of e. The pseudo-code of the MSB-first and the LSB-first binary algorithms are shown in Figures 6.1 and 6.2, respectively. The computational complexity of the algorithm in Figure 6.1 is given as,

$$P(e,m) = m + H(e) - 2 = \lfloor log_2(e) \rfloor + H(e) - 1 \tag{6.3}$$

Require: $\mathbf{x}, \mathbf{n}, e = (e_{m-1} \dots e_1 e_0)_2$.
Ensure: $\mathbf{y} = \mathbf{x}^e \bmod n$.
1: $\mathbf{y} = x$;
2: **for** $i = m-2$ downto 0 **do**
3: $\quad \mathbf{y} = \mathbf{y}^2$;
4: $\quad$ **if** $e_i == 1$ **then**
5: $\quad\quad \mathbf{y} = \mathbf{y} \cdot \mathbf{x}$;
6: $\quad$ **end if**
7: **end for**
8: **Return(y)**

Figure 6.1: MSB-First Binary Exponentiation

Require: $\mathbf{x}, \mathbf{n}, e = (e_{m-1} \dots e_1 e_0)_2$.
Ensure: $\mathbf{y} = \mathbf{x}^e \bmod n$.
1: $\mathbf{p} = \mathbf{x}$; $\mathbf{y} = 1$;
2: **for** $i = 0$ to $m-1$ **do**
3: $\quad$ **if** $e_i == 1$ **then**
4: $\quad\quad \mathbf{y} = \mathbf{y} \cdot \mathbf{p}$;
5: $\quad$ **end if**
6: $\quad \mathbf{p} = \mathbf{p}^2$;
7: **end for**
8: **Return(y)**

Figure 6.2: LSB-First Binary Exponentiation

An Example. Let us define e = 1903 = $(11101101111)_2$. Then m = 11 and $H(e)$ = 9. According to 6.2 the computational complexity of the binary algorithm is given as, $P(e) = m + H(e) - 2$= 11+9-2=18. After evaluating the algorithm of Figure 6.1, the resulting binary

$$x^1 \rightarrow x^2 \rightarrow x^3 \rightarrow x^6 \rightarrow x^7 \rightarrow x^{14} \rightarrow x^{28} \rightarrow x^{29} \rightarrow x^{58}$$
$$\rightarrow x^{59} \rightarrow x^{118} \rightarrow x^{236} \rightarrow x^{237} \rightarrow x^{474} \rightarrow x^{475} \rightarrow x^{950}$$
$$\rightarrow x^{951} \rightarrow x^{1902} \rightarrow x^{1903}.$$

Figure 6.3: Binary Exponentiation example

sequence is given as. We compare the MSB-first and the LSB-first binary algorithms in terms of time and space requirements below:

- Both methods require $m-1$ squarings and an average of $\frac{1}{2}(m-1)$ multipUcations.
- The MSB-first binary method requires two registers: x and y.
- The LSB-first binary method requires three registers: x, y, and P. However, we note that P can be used in place of M, if the value of M is not needed thereafter.
- The multiplication (Step 4) and squaring (Step 5) operations in the LSB first binary method are independent of one another, and thus these steps can be parallelized. Provided that we have two multipliers (one multipher and one squarer) available, the running time of the LSB-first binary method is bounded by the total time required for computing $h-1$ squaring operations on $k-bit$ integers.

We will consider the MSB first verion in the Montgomery Exponentiation technique for the current work.

6.2.2 Montgomery Exponentiation

The Montgomery product [32] algorithm is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute a modular exponentiation, i.e., the computation of $M^e \pmod n$. We replace the exponentiation operation by a series of square and multiplication operations modulo n. This is where the Montgomery product operation finds its best use. In the following we summarize the modular exponentiation operation which makes use of the Montgomery product function $MonPro$. The exponentiation algorithm uses the binary method [9].

function ModExp(M, e, n) { n is an odd number }
Step 1. Compute n' using the extended Euclidean algorithm.
Step 2. $\bar{M} := M \cdot r \bmod n$
Step 3. $\bar{x} := 1 \cdot r \bmod n$
Step 4. **for** $i = k - 1$ **down to** 0 **do**
Step 5. $\quad \bar{x} := \text{MonPro}(\bar{x}, \bar{x})$
Step 6. $\quad$ **if** $e_i = 1$ **then** $\bar{x} := \text{MonPro}(\bar{M}, \bar{x})$
Step 7. $x := \text{MonPro}(\bar{x}, 1)$
Step 8. **return** x

Figure 6.4: Montgomery Exponentiation Algorithm

Thus, we start with the ordinary residue M and obtain its n-residue M using a division like operation, which can be achieved, for example, by a series of shift and subtract operations. Additionally, Steps 2 and 3 require divisions. However, once the preprocessing has been completed, the inner-loop of the binary exponentiation method uses the Montgomery product operations which performs only multiplications modulo 2^k and divisions by 2^k. In this work, an inversion algorithm was presented which avoids divisions in the calculation of n'. When the binary method finishes, we obtain the n-residue x of the quantity $x = M^e \pmod{n}$. The ordinary residue number is obtained from the n-residue by executing the *MonPro* function with arguments x and 1. This is easily shown to be correct since

$$x = x \cdot r \pmod{n}$$

immediately implies that

$$x = x \cdot r^{-1} \pmod{n} = x \cdot 1 \cdot r^{-1} \pmod{n} := MonPro(x, 1). \tag{6.4}$$

The resulting algorithm is quite fast as was demonstrated by many researchers and engineers who have implemented it, for example, see [33]. However, this algorithm can be refined and made more efficient, particularly when the numbers involved are multi-precision integers. For example, Dussé and Kaliski [33] gave improved algorithms, including a simple and efficient method for computing n'. However, in this work, we will not include an interleaving multiplication technique, so as to increase throughput. This means that if the Montgomery Exponentiation takes in a unit size of 2^k bits, no additional splicing of the bits or serial shifting of the same occurs. The output is calculated immediately as the input becomes available. However, since the Montgomery Eponentiation and Montgomery product operations are sequential, they are subjective to the system clock's speeds for returning a result. However, the Karatsuba-Ofman multiplier implementation is a combinational one.

6.3 Montgomery Exponentiation implementation

The layout for Montgomery based exponentiation is given in the following figure 6.5.

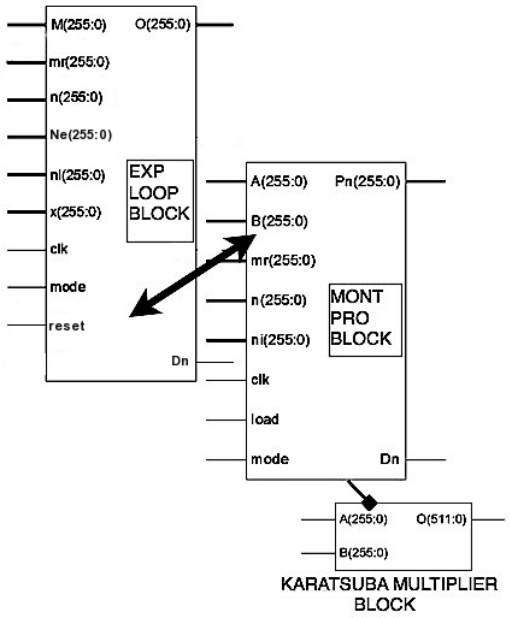

Figure 6.5: Montgomery Exponentiation Implementation Layout

The ports for each block is defined below:

EXP LOOP BLOCK

1. *M* takes in the base value to be exponentiated. It is a 256 bit value which is initially shifted by $r \pmod{n}$.
2. *mr* takes in the value of *r*-1'b1 which is Montgomery's shifting number , for implementing shifts inside the EXP LOOP BLOCK. This is usually a constant value. However in this work the value of *r* is taken as 2^256. This conserves the property mentioned in Chapter 5 for Montgomery product.
3. *n* is a 256 bit input which stores the modulus value.
4. *Ne* is the encryption or decryption exponent. It can take upto 256 bit input.
5. *ni* takes in the inverse of $-n \pmod{r}$ as in the Montgomery Algorithm.

6. x takes in the constant multiplier as seen in the Montgomery Exponentiation Algorithm. Its value is $1 \cdot r \pmod{n}$. Hence it requires pre-computation of modulus in the top module.

7. *clk* is used as a clock input from the system clock, because this implementation is partly sequential.

8. *mode* bit is a very crucial optimization. This avoids unnecessary hardware redundancy in the top module for the complete system. The *mode* signal is used to switch from modular exponentiation to direct Karatsuba multiplication as required by the RSA.

9. *reset* bit is used for resetting the values in the registers of the entire block.

10. *Out* is final 256 bit modular exponentiation result.

11. *Dn* secures the completion of all operations in the block.

The EXP LOOP BLOCK utilizes the clock signal to iterate an internal counter, which performs the function of the exponential loop seen in the above algorithms. The implementation also utilizes a bit shifting technique to obtain the current and next bits of the encryption/decryption exponent so as to perform the square and multiply operation. This operation has the disadvantage that it is directly dependent upon the clock, thereby making it susceptible to timing attacks [39].

MONT PRO BLOCK

This module performs the Montgomery product and gives out result in *Pn* port which is of 256 bits.
The sub-block KARATSUBA MULTIPLIER BLOCK intermittently provides results in a single clock cycle as mentioned in the synthesis results of the previous chapter.

load bit initializes the module for performing Montgomery Product.

KARATSUBA MULTIPLIER BLOCK

This block essentially implements the Karatsuba-Ofman multiplier that was discussed in the previous chapter.

6.3.1 Simulation Results

A sample simulation of the above is shown in fig 6.6. The inputs to the Montgomery Exponen-

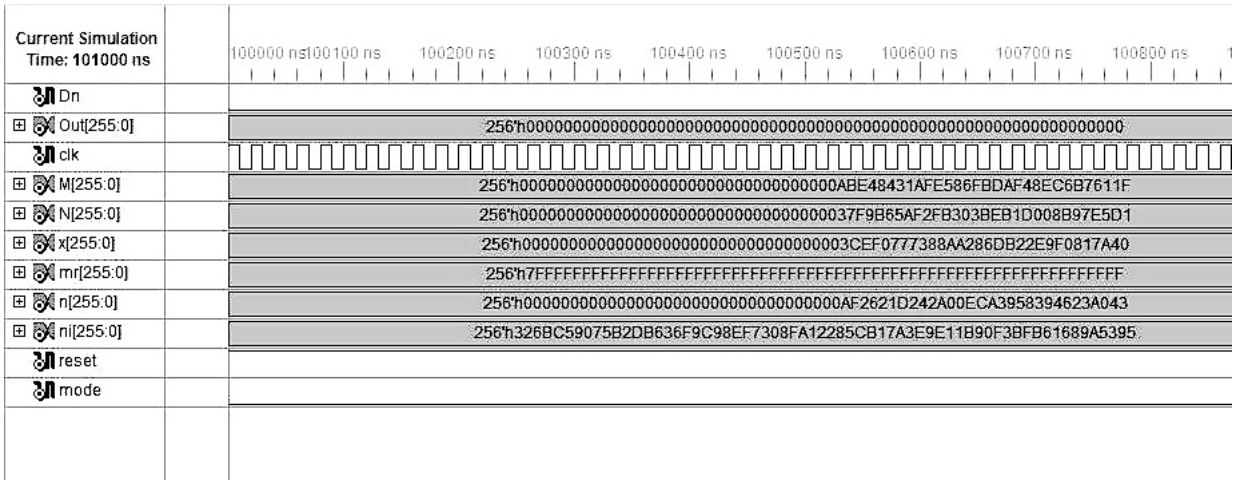

Figure 6.6: Montgomery Exponentiation Simulation

tiation Module used in this simulation are as follows:
M was initialized to abe48431afe586fbdaf48ec6b7611f in hexadecimal. The shifted value of 47f48d12669e2a2e53e0a5c3d2b4de in hexadecimal mod n was initially computed.
Ne was initialized to 37f9b65af2fb303beb1d008b97e5d1 in hexadecimal
mr was initialized to 7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff in hexadecimal
x was initialized to 3cef0777388aa286db22e9f0817a40.This is the shifted value of r mod n
n was initialized to af2621d242a00eca3958394623a043 in hexadecimal.
ni which is the negative inverse mod r was initialized to
326bc59075b2db636f9c98ef7308fa12285cb17a3e9e11b90f3bfb61689a5395 in hexadecimal
The output *Out* was obtained as a1124634758798086756746464764 in hexadecimal.

6.3.2 Synthesis Results

The synthesis results for the implementation of a modified Extended Euclidean with remainder result of upto 256 bits is shown below in the following table:

Implementation	*FPGA*	*Logic Slices* Used	*Bit* Length	*Frequency* (MHz)	*Throughput* Rate(bps)
Current work	Virtex II pro	9498	256	120.052	20.04M
(CSA based) by [40]	XC2V3000	11,304	512	102.31	5.1M
(CSA based) by [40]	XC2V6000	23,208	1024	95.9	4.79M
(CSA based) by [13]	XC2V3000	6294	512	168.38	9.28M
(CSA based) by [13]	XC2V6000	12537	1024	152.49	8.44M

Table 6.1: Synthesis Results for Montgomery Modular Exponentiation

Since, this implementation uses the $Karatsuba - Ofman$ technique discussed earlier, the same uses only a single clock cycle to generate a 512 bit product. However, the Montgomery Product step involves 3 calls to the *Karatsuba Multiplier* Block. Hence 3 clock cycles are required for computing the Montgomery product. Further, the Montgomery Exponentiation uses the MSB-First Binary Exponentiation technique. With a worst case scenario of a 256 bit exponent e, we have by 6.3 , $m = 256$, $H(e) = 256$ if all of the bits are 1s ; $P(e) = 510$. Hence there are 510 calls to the Montgomery product. However we must consider the final call to the Montgomery product. This operation is trivial , however it consumes 3 extra cycles. Therefore, the total number of cycles for the exponentiation operation can be calculated as (510 x 3)+3 = 1533 cycles, if the addition operations are considered to be trivial.

In, the above table, the current results are compared with 512 bit and 1024 bit implementations of RSA [40, 13] . It is evident from the above table that the present implementation gives a better throughput. However, the slice consumption for this implementation was comparatively higher and the bit lengths were very low in this work, which makes the comparison difficult to estimate the results. Further, the Maximum frequency reported by Xilinx ISE was 120 Mhz, which saturates the 100 Mhz bus frequency of the Virtex 2 pro board. Hence, the maximum frequency possible for this implementation was limited by the available $FPGA$.

6.4 Conclusion

Different techniques for exponentiation were discussed, out of which, the Montgomery Exponentiation was chosen as the modular exponentiation techniques because it is a very fast algorithm. The synthesis results show a definitive improvement in the throughputs , however , due to limitations of hardware, it was not possible to increase the size of the modulus for comparison. Further, the layout of the current implementation was presented along with the synthesis and simulation results.

Chapter 7

RSA Implementation and Synthesis Results

7.1 Introduction

This thesis presented some of most efficient techniques available for creating and developing a cryptosystem for modular arithmetic. Barring some of the initial conditions such as Random Number Generation and Primality testing, an RSA modular crypto engine which is generic upto a 256 bit modulus was synthesized. The results of this synthesis is presented in this chapter. The RSA may be implemented both as a low throughput system with less number of Logic Slices or as a high throughput system with larger number of slices. The latter method was preferred for this implementation because of a greater attention to speed. The objective of a generic RSA with upto a 256 bit modulus was thereby achieved.

However, it must be reminded that a Virtex II pro FPGA with speed grade -7 on a Digilent board was utilized for this implementation. This is because the Virtex II pro FPGA contains 13,696 available slices which accounts to $\approx$ 3424 CLBs (Configurable Logic Blocks) out of which 90 % was utilized(12,374 slices). Hence $\approx$ 3094 CLBs were utilized in this process. The Layout for this implementation was presented in Chapter 2.

7.1.1 Simulation Results

A sample simulation of the RSA implementation is shown in figure 7.1

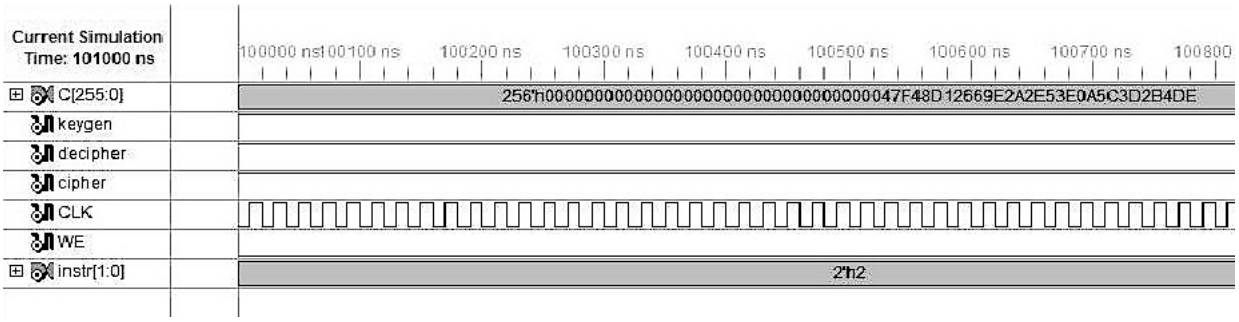

Figure 7.1: RSA Implementation Simulation

The inputs to the RSA Engine were stored in a register block of size 3x256 bits.
The values stored are listed corresponding to the RSA algorithm equivalents.The primary keys p & q were 128 bit registers, which utilized a third of the register block.
The value of p was 113680897410347 in decimal.
The value of q was 7999808077935876437321 in decimal.

The value of encryption component e is also initialized in this register block. It can be upto 256 bits. The value of e was 97 in decimal.
The port $instr$ was used for a manual control of the Key Generation, Encryption and Decryption process. Additionally, if the $instr$ is given the value zero for resetting the complete cryptosystem.
The bits for $keygen, cipher\ and\ decipher$ indicate completion of key generation,encryption and decryption process.
The WE bit was used for re-initializing the register block with the default values.
The value of plaintext P was a1124634758798086756746464764 in hexadecimal.
The resulting cipher C was obtained as 47f48d12669e2a2e53e0a5c3d2b4de in hexadecimal.

7.1.2 Synthesis Results

The synthesis results for the implementation of a modified Extended Euclidean with remainder result of upto 256 bits is shown below in the following table:

Logic Component	*Used*	*Available*
Number of Slices	12,374	13,696
Number of Slice Flip Flops	4,121	27,392
Number of 4 input LUTs	23,810	27,392
Number of bonded IOBs	263	556
Number of Multipliers 18x18	81	136

Table 7.1: Synthesis Results for the RSA cryptosystem

The value of the plaintext was initialized prior to implementation. The input range can be upto 255 bits (adjusting for modulus). Alternatively an input port of 256 bit width can also be specified for this purpose.

The key-generation step initially performs two multiplication operations each. Each such operation takes 1 clock cycle in this implementation.The next operation performs a non-restoring division operation and an inversion operation in parallel, since both modules were implemented separately. The results were used for Montgomery exponentiation in the next stage of encryption and subsequent decryption. Both modules consume 512 clock cycles in the worst case.

The encryption and decryption operations takes one initial remainder calculation and an exponentiation operation to complete.Thus, the encryption and decryption operations are the most relavant among all other operations (1533 cycles).

Throughout the implementation, the register block was used for primary key, decryption and plaintext storage. During synthesis , this block was converted to registers , which contributed to the overall slice count.

7.2 Extensions to this work

Since the possiblity of this cryptosystem being used as a subsystem for implementation in an Embedded system is imminent, the portability to EDK was also considered. The EDK provided a complete set of ip cores which allows for adding or modifying the specifications of each device that can be used on the Virtex 2 pro board.The procedure starts of with a base system build that includes options for including a soft core or hard core processor.The preference was give to a hard core processor, so that the design would be a standard.The Xilinx EDK offers the *Microblaze* processor as their solution of a soft core.The Virtex 2 pro can be configured to have upto 2 hard core processors that is based on the Power PC *PPC 405*.

The base system offered a serial interface via RS232 and a 100 Mbps MAC PHY ethernet interface.The serial interface was chosen to setup a mini console to start or edit settings of the implementation.This left, only the ethernet interface, as an option to be used as input to the cryptographic algorithm. However, due to the sheer size of this implementation, only a webserver was set up as initial interface to this project in Xilinx based embedded C. This is still a work in progress. Hence, no results have been included in this thesis work.

7.3 Conclusion

This thesis has summarized the work which was initiated during the summer of 2008.This report has also summarized some of the key techniques required for RSA based crytosystems. Finally, the report concludes with the RSA implementation simulation and synthesis results. The results in each phase of the implementation were covered and presented in this thesis.

Bibliography

[1] F. Rodríguez-Henríquez, N. Saquib, A. D. Pérez, and Çetin Kaya Koç, *Cryptographic Algorithms on Reconfigurable Hardware*, ser. Signals and Communication Technology. Springer, 2007, vol. XXVI.

[2] J. Fry and M. Langhammer, "Fpgas lower costs for rsa cryptography." [Online]. Available: http://www.design-reuse.com/articles/6358/fpgas-lower-costs-for-rsa-cryptography.html

[3] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *English Translation in Soviet Physics Doklady*, vol. 7, pp. 595–596, 1963.

[4] "Tpm specification version 1.2 revision 103:part 1, design principles," 2007. [Online]. Available: http://www.trustedcomputinggroup.org/resources/tpm_specification_version_12_revision_103_part_1__3

[5] I. C.E. and L. K., "Trusted hardware: Can it be trustworthy ?" *Design Automation Conference. DAC '07. 44th ACM/IEEE*, pp. 1–4, June 2007.

[6] L. Hars, "Modular inverse algorithms without multiplications for cryptographic applications," *EURASIP Journal on Embedded System*, vol. 2006, January 2006.

[7] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, February 1978.

[8] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, 2nd ed. Addison-Wesley, 1981, vol. 2.

[9] Çetin Kaya Koç, "High-speed rsa implementation," RSA Laboratories, Redwood City, CA,, Tech. Rep. TR 201, 1994.

[10] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, July 1985.

[11] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, pp. 644–654, November 1976.

[12] N. I. for Standards and Technology, "Digital signature standard (dss)," August 1991.

[13] M. D. Shieh, J. H. Chen, H. H. Wu, and W. C. Lin, "A new modular exponentiation architecture for efficient design of rsa cryptosystem," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems archive*, vol. 16, no. 9, pp. 1151–1161, September 2008.

[14] (2009, May) Ubasic, version 8.74. [Online]. Available: http://archives.math.utk.edu/software/msdos/number.theory/ubasic/.html

[15] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in gf(2^m) using normal basis." *Information and Computing*, vol. 78, pp. 171–177, 1988.

[16] K. Fong, D. Hankerson, J. Lopez, and A. Menezes, "Field inversion and point halving revisited," *IEEE Trans. Computers*, vol. 53, no. 8, pp. 1047–1059, 2004.

[17] N. Takagi, J. Yoshiki, and K. Tagaki, "A fast algorithm for multiplicative inversion in gf(2^m) using normal basis," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 394–398, May 2001.

[18] M. A. Hasan, "Efficient computation of multiplicative inverses for cryptographic applications," in *15th IEEE Symposium on Computer Arithmetic*. Vail,Colorado, U.S.A.: IEEE, 2001.

[19] A. A. A. Gutub, A. F. Tenca, E. Savas, and Q. K. Kog, "Scalable and unified hardware to compute montgomery inverse in gf(p) and gf(2^n)," in *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop*, vol. 2523. Redwood Shores, CA, USA, August 2002, pp. 484–499.

[20] F. R. Henríquez, N. A. Saqib, and N. CruzCortés, "A fast implementation of multiplicative inversion over gf(2^m)," *International Conference on Information Technology: Coding and Computing. ITCC 2005*, vol. 1, pp. 574–579, April 2005.

[21] J. Guajardo and C. Paar, "Itoh-tsujii inversion in standard basis and its application in cryptography and codes," *Designs, Codes and Cryptography*, vol. 25, pp. 207–216, 2002.

[22] R. Schroeppel, H. Orman, S. W. O'Malley, and O. Spatscheck., "Fast key exchange with elliptic curve systems," in *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '95. London, UK: Springer-Verlag, 1995, pp. 43–56.

[23] M. N. Cervantes, K. G. Avila, and F. R. Henríquez, "Investigating modular inversion in binary finite fields," Computer Science Department CINVESTAV-IPN, Mexico, Tech. Rep. 2006-1, May 2006.

[24] F. R. Henríquez, G. Marales-Luna, N. A. Saquib, and N. Cruz-Cortés, "Parallel itoh-tsujii multiplicative inversion algorithm for a special class of trinomials," *Designs, Codes and Cryptography*, vol. 45, no. 1, pp. 19–37, October 2007.

[25] H. Sedlak, "The rsa cryptography processor," in *Advances in Cryptology - EUROCRYPT '87*, D. Chaum and W. L. Price, Eds. Berlin, Germany: Springer-Verlag, 1987, vol. 304, pp. 95–105.

[26] P. Barret, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology - CRYPTO '86*. Berlin, Germany: Springer-Verlag, 1986, vol. 263, pp. 311–323.

[27] A. J. Menezes, V. O. P. C., and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, Florida, USA: CRC Press, 1997.

[28] E. F. Brickell, "A fast modular multiplication algorithm with applications to two key cryptography," in *Advances in Cryptology - CRYPTO '82*, R. L. R. D. Chaum and A. T. Sherman, Eds. New York, USA: Plenum Publishing, 1982, pp. 51–60.

[29] J. K. Omura, "A public key cell design for smart card chips." *n International Symposium on Information Theory and its Applications*, pp. 983–985, 1990.

[30] J. Quisquater, "Encoding system according to the so-called rsa method, by means of a microcontroller and arrangement implementing this system," *United States Patent,Patent Number 5166978*, 1992.

[31] D. De Waleffe and J. Quisquater, "Corsair: A smart card for public key cryptosystems," in *Advances in Cryptology - CRYPTO '90*, A. J. Menezes and S. A. Vanstone, Eds. Berlin: Springer-Verlag, 1990, vol. 537, pp. 502–514.

[32] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[33] S. R. Dusse and K. B. S., "A cryptographic library for the motorola dsp56000," in *Advances in Cryptology - EUROCRYPT '90*, I. B. Damgard, Ed. Berlin, German: Springer-Verlag, 1990, vol. 473, pp. 230–244.

[34] S. Eldridge and C. Walter, "Hardware implementation of montgomery's modular multiplication algorithm," *IEEE Transactions on Computers*, vol. 42, no. 6, pp. 693–699, June 1993.

[35] A. Mazzeo, L. Romano, G. P. Saggese, and N. Mazzocca, "Fpga-based implementation of a serial rsa processor," in *Proceedings of the conference on Design, Automation and Test in Europe*, vol. 1. Munich, Germany: IEEE Computer Society, March 2003, p. 10582.

[36] V. Serrano-Hernández and F. Rodríguez-Henríquez, "An fpga evaluation of karatusba-ofman multiplier variants," Computer Science Department CINVESTAVIPN, Mexico, Technical Report CINVESTAV_COMP 2006 2, May 2006.

[37] J. Gathen and J. Shokrollahi, "Efficient fpga based karatsuba multipliers for polynomials over f_2," in *Revised Selected Papers,Lecture Notes in Computer Science*, vol. 3897. Kingston,ON, Canada: Springer-Verlag, 2006, pp. 359–369.

[38] C. Grabbe, M. Bednara, J. Teich, J. Gathen, and J. Shokrollahi., "Fpga designs of parallel high performance gf(2^{233}) multipliers," in *Proceedings of the 2003 International Symposium on Circuits and Systems,ISCAS '03*, vol. 2, May 2003, pp. 268–271.

[39] E. Oksuzoglu and E. Savas, "Parametric, secure and compact implementation of rsa on fpga," *International Conference on Reconfigurable Computing and FPGAs*, December 2008.

[40] C. McIvor, M. McLoone, and J. McCanny, "Modified montgomery modular multiplication and rsa exponentiation techniques," *IEE Proceedings in Computers and Digital Techniques*, vol. 151, no. 6, pp. 402–408, November 2004.